Software Security: Vulnerabilities and Countermeasures for Two Attacker Models

Frank Piessens
iMinds-DistriNet
Katholieke Universiteit Leuven
Belgium
Email: Frank.Piessens@cs.kuleuven.be

Ingrid Verbauwhede
iMinds-COSIC
Katholieke Universiteit Leuven
Belgium
Email: Ingrid.Verbauwhede@esat.kuleuven.be

***Abstract*—History has shown that attacks against network-connected software based systems are common and dangerous. An important fraction of these attacks exploit implementation details of the software based system. These attacks – sometimes called *low-level* attacks – rely on characteristics of the hardware, compiler or operating system used to execute software programs to make these programs misbehave, or to extract sensitive information from them. With the increased Internet-connectivity of embedded devices, including industrial control systems, sensors as well as consumer devices, there is a substantial risk that similar attacks will target these devices.**

This tutorial paper explains the vulnerabilities, attacks and countermeasures relevant for low-level software security. The paper discusses software security for two different attacker models: the classic model of an attacker that can only interact with the program by providing input and reading output, and the more recent and challenging model of an attacker that controls part of the execution platform on which the software runs, for instance because the attacker has compromised the operating system, or some of the libraries that the software under attack relies on.

I. Introduction

Security is about maintaining desirable properties of systems in the presence of intelligent adversaries. Hence, to define a security problem, one must define (1) the system under consideration, (2) the desirable properties that one wishes to maintain (the *security objective*), and (3) the capabilities the attacker or adversary is assumed to have (the *attacker model*).

Software security, obviously, studies the case where the system is a software based system. History has shown that software based systems, and in particular systems that are connected to the Internet, are vulnerable to a wide variety of attacks. Attacks against software can take many forms, but this paper focuses specifically on attacks that exploit implementation details of the platform (i.e. the hardware and software infrastructure) on which the software is executing. These attacks range from the classic stack-smashing attack [1] to modern attack techniques like Return-Oriented-Programming (ROP) attacks [2] or memory scanning malware [3].

This class of attacks (sometimes dubbed *low-level software attacks*) has been one of the most damaging classes of attacks on the Internet over the past decades. With the increased Internet-connectivity of embedded devices, including industrial control systems, sensors as well as consumer devices, there is a substantial risk that similar attacks will also be launched against such devices.

In this paper, we consider software systems that are compiled from source code, and we consider a very general security objective: *the compiled system should behave as specified in the source code that it is compiled from* (and *only* as specified in the source code, i.e. no additional unexpected functionality). This security objective defends against a very wide range of attacks that exploit platform implementation details, including all the example attacks mentioned above. For instance, the behaviour of a vulnerable program under a stack-smashing attack diverges completely from what is specified in the source code of the program.

We can study this security objective for compiled software systems under different attacker models. The most widely studied attacker model considers an attacker that can provide input to, and read output from a compiled program. The attacker's goal is to choose the inputs in such a way that behaviour of the running program deviates from the behaviour specified in the source code. Of course, attackers often have a very specific deviating behaviour in mind (like getting a root shell on the computer under attack, or installing a root-kit on that computer) but from a defense point of view, it makes sense to consider any behaviour that deviates from what is specified in the source code as an attack.

Such attacks are possible in this attacker model against software that is written in unsafe languages like C or C++ and that contains so-called *memory safety vulnerabilities*. We discuss those vulnerabilities in Section III, as well as the wide range of attacks and countermeasures that has been studied in the past decades for this attacker model.

In some cases however, a more powerful attacker model is appropriate. In Section IV, we consider attackers that can provide part of the compiled code of a program. This attacker model is relevant for software that consists of multiple modules or components that are compiled separately and then linked together, and loaded to be executed. Such software often includes compiled modules from third parties, and in this attacker model we consider attackers that can arbitrarily modify one or more of these modules. The model also includes attackers that can modify code (for instance to install malware) in the more privileged operating system layer.

We assume that readers of this paper have programming experience in an imperative programming language like C, and have a general understanding of how such languages are compiled to a standard von Neumann style computer that executes low-level, unstructured machine code. In the next Section, we briefly recap some details of this compilation process that are important to understand the attacks and countermeasures in this paper, but this is *not* intended to be a self-contained introduction to computer architecture and compilation. Readers who need to refresh that background more extensively should consult a relevant textbook [4].

II. Source code, machine code and compilation

Most software is developed as source code in a high-level programming language and subsequently compiled to machine code for execution. The difference between source code and machine code is substantial. In high-level languages like C, control flow is structured, there is a strict separation between code and data, and most languages support features that allow developers to define and enforce abstractions or to hide information behind interfaces. At machine code level, there is a single virtual address space, where both code and data are represented as binary n-bit words and where control flow is unstructured. For the examples in the paper, we will assume a 32-bit memory address space, i.e. virtual memory consists of 2^{32} bytes. The processor has 32-bit registers and a single instruction can load a 32-bit word (i.e. 4 bytes) from memory to a register.

Figure 1 illustrates the difference between source and machine code, and also illustrates some details of the compilation process.

Part (a) of the Figure shows a very simple C program that is intended to be representative of a server process. In the main method, it should first initialize, listen on a network socket and accept connections (code not shown), and then it calls the process () method passing it a file descriptor that can be used to communicate on the established connection. The process() method uses a get_request() method to read a request of the connection.

Part (b) of the Figure shows the compiled code for the process() method. Both the assembly code, as well as the machine code (in hexadecimal notation) are shown, and comments clarify what the code is doing. In particular, the code shows how the necessary management information associated with this specific function invocation is maintained on the call stack. The base pointer is a processor register that points to the base of the stack record associated with the current function invocation. The machine code of process() starts with saving the old base pointer (the one associated with the main() function invocation), and then sets the new base pointer to the top of the stack. Next, it allocates space on the stack by subtracting 0x18 from the stack pointer (note that this grows the stack, as the stack grows towards lower memory addresses). This includes space for the local variable (buf), as well as space for the two parameters that will be passed to get_request() in the first statement of process(). The two parameters are copied into that allocated space, and then the call to get_process() happens.

Part (c) of the figure shows a snapshot of (parts of) the run-time state of the process executing the compiled program at the point where it has just entered the get_request() function. You can recognize the stack records (also called *activation records*) of the process() function invocation and of the get_request() function invocation. You can also see the machine code of the process() function at some other place in memory: it is stored starting at address 0x080483f2, in little-endian byte order (i.e. the first byte is stored in the least significant byte of the word at address 0x080483f2).

Two of the processor registers – the Instruction Pointer (IP) and the Stack Pointer (SP) – are also shown, and point to code of the get_request() function and the top of the stack record for the get_request() invocation (which is at this point in time the top activation record on the stack) respectively.

While not all details in the Figure are important, the Figure does illustrate the huge abstraction gap between source code and machine code. At run-time, the entire state of the executing program (including its code, its data and information about what point of execution the program has reached) is maintained in memory and in processor registers; the run-time state is essentially just a large collection of 32-bit words.

Depending on many factors, including at least the precise characteristics of the source language, the way in which the source code is compiled to machine code and the way in which attackers can interact with the compiled code, attackers can exploit the characteristics of the machine code level to make software misbehave in dangerous ways.

We will illustrate some of these potential exploits in the next Sections, referring back to Figure 1 for some of the examples.

III. Low-level software security in the I/O attacker model

In the I/O attacker model, the attacker can only provide input to, and observe output of the program. This is an appropriate model, for instance, for attackers interacting with server software over a network connection, and these classes of attacks have been among the most important security issues for networked software for several decades.

A program is vulnerable under this attacker model only if it contains memory safety vulnerabilities: bugs that may cause the program to write to memory cells not allocated to the program. We study these vulnerabilities in Subsection III-A. The presence of these vulnerabilities enables a wide range of attack techniques, some of which we explore in Subsection III-B. Countermeasures for such attacks have been studied for decades, and we provide a brief survey of important countermeasures in Subsection III-C.

A. Memory safety vulnerabilities

Source languages like C support mutable state, i.e. the source language has constructs for allocating and deallocating memory that can subsequently be assigned to, or read from.

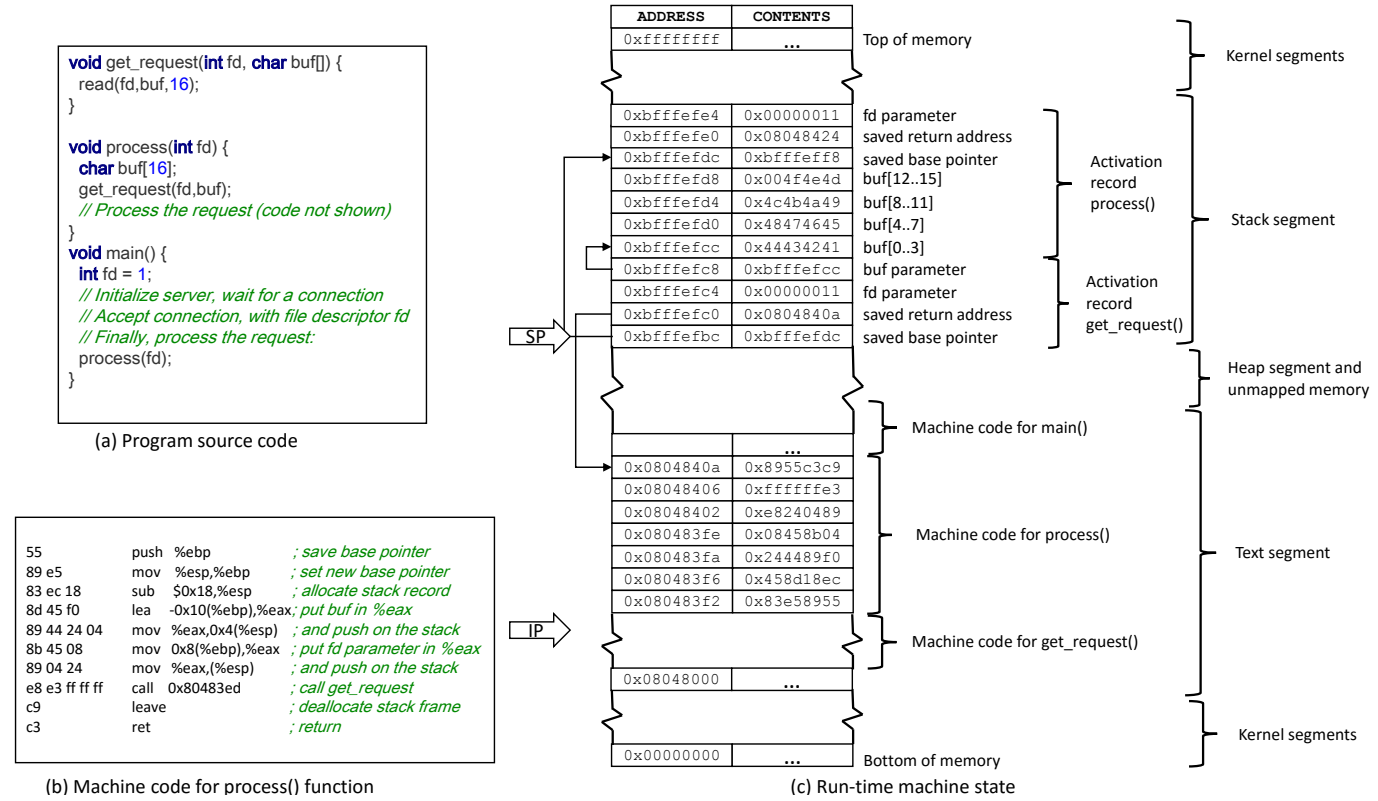

Fig. 1. This figure illustrates the relationship between source code and run-time machine state.
Part (a) shows a simple source code program.
Part (b) shows the compiled version of one of the functions in that program. Both the machine code (in hexadecimal) as well as the corresponding assembly code is shown. Note for instance that assembly code instructions have variable lengths: they are between 1 and 5 bytes long.
Part (c) shows a snapshot of the run-time machine state while this program is executing. It shows parts of the memory address space, as well as the Instruction Pointer (IP) and Stack Pointer (SP), at the point when the program has just entered the get_request() function. Note that the machine code shown in part (b) is stored in memory using a little-endian byte order.

At run-time, on the machine code level, these memory cells allocated for use in the program will be part of the same virtual address space where also program code, and management information to track the structured control flow will be stored. For example, in Figure 1, the local variable buf of the process() function is allocated at address 0xbfffefcc and occupies 4 words (each containing 4 bytes). In the snapshot of memory shown in Figure 1, the buf variable contains the string "ABCDEFGHIJKLMNO": the ASCII codes of these characters are stored (in little-endian order) in the 4 words allocated to buf, ending with a null byte.

In the presence of bugs in the source program, it is possible that the program writes to memory cells that are not allocated for use in the program, and in certain circumstances this can cause the program to modify program code or management information in dangerous ways. For instance, in Figure 1, we can introduce such a bug in the get_request() function by replacing the third parameter of the read call with 32 instead of 16. Now, input provided to the program might overflow the space allocated to the buf variable by 16 bytes (4 words), and hence it can for instance overwrite the saved return address stored at address 0xbfffefe0.

A *memory safety vulnerability* is a bug where a program reads or writes to a memory cell not currently allocated to the program. In C-like languages, these bugs come in two forms.

A program has a *spatial* vulnerability, if it accesses a range of cells (typically an array) that is allocated to the program, but due to insufficient or buggy checking these accesses might go out of bounds. The example above is an example of a spatial memory safety vulnerability. The term *buffer overflow* is often used as a synonym for such spatial vulnerabilities.

A program has a *temporal* vulnerability, if the program accesses a cell that was once allocated to the program, but has since been deallocated. Such deallocation can happen implicitly or explicitly. If for instance the process() function in Figure 1 were to return buf to the main() function, and the main() function would access buf, for instance by reading input into buf, this would be an example of a temporal vulnerability: since it is a local variable, the buf array is only valid during the invocation of process() and is deallocated (implicitly, by deallocation of the corresponding stack activation record) on return of process().

The range of memory cells the attacker can illegitimately access depends on the vulnerability. For instance, in the spatial vulnerability example above, the attacker can modify 16 bytes (4 words) of memory outside of the buffer's allocated memory

space, more specifically the words with addresses 0xbfffefdc – 0xbfffefe8. In a vulnerability where the program performs a buf[i] = v assignment, where both i and v come from input channels and hence can be controlled by the attacker, the range of cells modifiable by the attacker is essentially the entire virtual address space (taking into account that this kind of indexing will wrap around when the top of memory is reached). The attacker can modify the contents at one address of his choosing in the entire memory address range.

B. Attack techniques

An attacker will try to find input values to send to the program such that a memory safety vulnerability is triggered. The behavior of a program in which such a vulnerability has been triggered is *undefined* according to the C language specification. In other words, the source program does not give any information anymore on how the program will behave from that point onward. In practice, what happens when a program accesses memory out of bounds, or memory that has been deallocated, depends on low-level details of the compiler, operating system and/or hardware. Often, the program will just crash.

The attacker should use his knowledge about the low-level details of the executing program to make the program do more useful things than just crashing. The oldest and most widely known technique is *stack smashing* with direct code injection [1]. This attack exploits spatial vulnerabilities on stack allocated buffers, like the example vulnerability discussed above. For that example, the attack would essentially go as follows: the attacker provides input that get_request() stores into buf, and by providing more than 16 bytes the attacker overwrites first the saved base pointer and then the saved return address. When the get_request() function, and subsequently the process() function return, this modified return address will be popped from the stack into the Instruction Pointer register, and the processor continues execution at this address of the attacker's choosing. The attacker will hence set the return address to such a value that the processor starts executing code that the attacker wants to execute. He can for instance set the return address to 0xbfffefcc, the address of buf. Then the processor will start loading bytes from that address, interpreting them as instructions, and executing them. Since the attacker provides these bytes as input, he can essentially choose what code the processor will execute. This technique, where the attacker brings machine code into memory as data is called *direct code injection*.

Over the past decades, a wide range of attack techniques has been developed. Some important examples include:

- *Overwriting code pointers*: the attacker overwrites a memory cell that will at some later point be loaded in the Instruction Pointer register. Examples of such memory cells include the saved return addresses in activation records on the stack (as discussed above), or memory cells that contain function pointers. For instance, a sorting function can take as a parameter a pointer to the comparison function that should be used for sorting. If the attacker can modify the pointer to the comparison function, control flow will be hijacked on the point where the comparison function is called through the function pointer.
- *Code corruption attacks*: instead of overwriting a code pointer, the attacker can overwrite the machine code of a part of the program that will later be executed. For example, referring to Figure 1 again, the attacker could overwrite the bytes starting at 0x0804840a, where execution will continue after return of the get_request() function. (Obviously, the attacker would need a vulnerability that gives him a range of accessible cells that includes these cells).
- *Code reuse attacks*: instead of redirecting control flow to a location in data memory as in the direct code injection example above, the attacker can redirect the control flow to existing code in code memory. Several variations of this technique exist: in a *return-to-libc* attack, the attacker will divert control flow to an existing useful function (typically a function defined in the libc library). In a *Return-Oriented-Programming (ROP)* attack [2], the attacker will divert control flow to a code fragment (called a *trampoline*) that (1) resets the Stack Pointer (SP) to a memory address whose contents is controlled by the attacker, and (2) returns. On this return, control flow will continue to the address that the SP points to (and hence is under control of the attacker). The attacker chooses this to be an address to another code fragment (called a *gadget*) that ends with a return instruction. Where this return will go is again chosen by the attacker, and by continuing this process the attacker can execute a sequence of such gadgets. It has been shown that by combining such gadgets, the attacker can essentially do anything he wants. These code reuse attacks are particularly useful in cases where there are countermeasures active that prevent the attacker from doing code corruption or from injecting code as data (direct code injection).
- *Data-only attacks*: instead of overwriting code pointers or code, the attacker can choose to overwrite the contents of another mutable variable in the program. A typical example would be to overwrite a boolean variable (e.g. isAdmin) that impacts the actions that the program under attack can perform. By modifying through a memory safety vulnerability the variable isAdmin from false to true, the attacker is now allowed to perform administrative actions.
- *Information leaks:* all examples we have discussed so far violate the integrity of certain memory cells. But also confidentiality attacks are possible. By *reading* past the bounds of a buffer, the program might leak confidential information such as cryptographic keys (this is essentially what was possible with the famous Heartbleed vulnerability). In addition, leaking the contents of parts of memory may allow the attacker to bypass some of the countermeasure we discuss further in this paper [5].

All these attack techniques are well understood and well documented. Erlingsson et al. [6] discuss detailed examples of many of these attack techniques, and Szekeres et al. [7] develop and describe a general model of these memory corruption attacks.

C. Countermeasures

Because of the widespread nature of the attacks discussed in the previous Subsection, there has been a substantial amount of research on countermeasures [8], [7], and several of these countermeasures are now widely adopted in practice.

We distinguish between countermeasures that counter *exploitation* of vulnerabilities during execution of the software (for instance by making some of the attack techniques discussed above substantially harder), and countermeasures that counter the *introduction* of memory safety vulnerabilities during the development or testing of the software.

1) Countering exploitation of vulnerabilities: The following three countermeasures are now widely adopted in most server and desktop software platforms, including the Windows platform and most Unix variants.

- *Stack canaries* [9] are a cheap and straightforward countermeasure against stack smashing: the compiler emits code to (1) place a (for the attacker) unpredictable value – the *canary* – in each activation record between the local variables and the saved registers (base pointer and stack pointer), and (2) checks that this value is not modified before returning from a function call. If an attacker were to overwrite a saved return address by overflowing a stack allocated buffer, he also necessarily needs to overwrite the canary, and hence this will be detected before the overwritten return address is used.
- *Data Execution Prevention (DEP)*: marking the code segment non-writable and the data segment non-executable is a simple countermeasure against direct code injection: the attacker can no longer bring data in memory and then later have it executed as code.
- Finally, *Address Space Layout Randomization (ASLR)*, introduces artificial randomness into the memory layout of a process, making it harder for an attacker to (1) predict interesting memory locations (like the location of a saved return address) to overwrite, and (2) write a (for the attacker) useful value – like the address of a trampoline or gadget to jump to – to these memory locations.

While the combination of these countermeasures raises the bar for attackers, it is commonly accepted that many memory safety vulnerabilities remain exploitable through clever combinations of attack techniques.

2) Countering introduction of vulnerabilities: Instead of countering the exploitation of memory safety vulnerabilities, it is even better to make sure that they are not present. A wide range of techniques exists to assist developers in writing safe code. Three important classes of techniques are:

- The use of *safe languages*, like Java or C#. These languages are designed such that a combination of compiler-enforced bounds checks (to avoid spatial vulnerabilities) and automated garbage collection (to avoid temporal vulnerabilities) can provide hard assurance that no memory safety vulnerabilities are present. There is currently a growing interest in more C-like languages that compile to machine code and give the programmer more control over memory management, while still providing substantial memory safety guarantees through type checking. A prototypical example is the Rust programming language [10], strongly influenced by the Cyclone language [11].
- If switching to a safe language is not an option, best-practice dictates a combination of *coding guidelines* [12], and *code review*. Source code analysis tools can help during code review. Some tools require little developer effort, but suffer from false positives and false negatives [13], other tools can provide very high assurance about memory safety but require a substantial amount of effort [14], [15].
- *Testing* for the presence of memory safety vulnerabilities is made significantly more effective with the use of run-time checks that check for unsafe memory accesses [16], [17]. While such run-time checks often impose a performance overhead that is unacceptable in production systems, this overhead can be acceptable during testing, and they help ensure that every illegal memory access is detected during test runs.

Again, none of these techniques is a silver bullet. Code review and testing can suffer from false negatives: even well reviewed and well tested code can still contain bugs. And even safe languages often link to libraries written in an unsafe language, or allow the programmer to write potentially unsafe code in well marked parts of the programs (for instance, unsafe blocks in C# or Rust).

IV. Low-level software security in the machine code attacker model

In the I/O attacker model, software without memory safety vulnerabilities is immune to low-level attacks. A safe program will behave as specified in the source code for all possible inputs an attacker can provide. But stronger attackers can still derail the program.

A stronger and realistic attacker model considers attackers that can install some machine code on the computer executing the program. This could be machine code within the virtual address space of the process executing the program (for instance because the program links to a native library controlled by the attacker, or because it loads a binary plugin provided by the attacker). It could also be machine code in the operating system kernel (for instance because the attacker succeeded in installing malware on the computer). A practical example of this kind of attack that is gaining importance is so-called *memory-scanning malware* – malware that will read the virtual address space of running processes looking for interesting data like credit-card numbers or passwords [3].

This attacker model is an appropriate model in cases where software is built using third-party libraries that might be malicious (or vulnerable, and hence might turn malicious

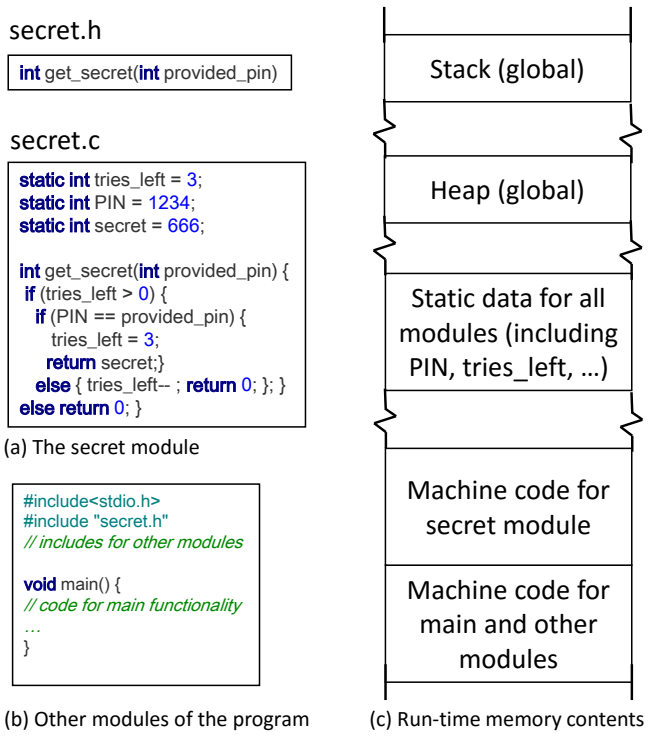

Fig. 2. A program with a security critical module.

after an exploit), or in cases where software might run on a compromised platform (like on a malware infected operating system).

As a small representative example, consider the program in Figure 2. The program has a single module (implemented in secret.c) that manages sensitive information, in this case the secret variable that should only be shown to users of the program who can provide a correct PIN. After three tries with an incorrect PIN, the module will refuse further attempts to protect against brute force attacks. The secret module exposes (in it's header file secret.h) only the get_secret() function, and hence access to the global variables in secret.c is restricted to the secret module at source code level. Other modules (including for instance the main() function) can only interact with the module through the get_secret() function. This module is a very simple example of a security critical module, like for instance the password manager in a browser or the implementation of a cryptographic protocol. These modules manage secrets, and even if they implement restrictions on access to these secrets in source code (like the small example module), they can still be subject to memory scraping attacks.

To see this, consider the compiled version of this program at run-time. Part (c) of Figure 2 shows a schematic picture of the memory contents at run-time. If we now consider an attacker that can choose the machine code for some of the other modules of the program, it is clear that the attacker can easily violate both integrity as well as confidentiality of the variables of the secret module. At machine code level, there is no restriction in place for the attacker's code to access the memory cells containing the secret module's variables. A memory scraping attack is an attack where such malicious code scans the entire virtual memory address space of a process looking for secrets like authentication credentials, cryptographic keys, credit card information and so forth. Even if the source code implements restrictions on accessing these secrets (like in the example secret module), a machine code attacker still can easily access them.

Note an important difference with the I/O attacker: the I/O attacker could only launch attacks in the presence of bugs (vulnerabilities) in the program, whereas even a bug-free program is still vulnerable to the machine code attacker.

In this Section, we consider how to protect modules like the secret module in this example against machine code attackers. This requires at least some form of isolation mechanism to protect parts of the virtual address space of a process. We discuss some of the proposed mechanisms in Subsection IV-A. Then we discuss in Subsection IV-B how the compiler should make use of these isolation mechanisms to guarantee security against low-level attacks, i.e. attacks that make the protected module behave in any way that deviates from what is specified in the source code.

A. Isolation mechanisms

Looking back at Figure 2, it is clear that some run-time support for isolating the various modules in a single address space is required to offer protection against the machine code attacker. A wide variety of such isolation mechanisms have been studied. Some important representative mechanisms are:

- *Virtual machines* like the Java Virtual Machine [18] raise the level of abstraction of compiled code such that it gets closer to that of the source code. Compiled modules now consist of bytecode, and both the distinction between data and code, as well as abstraction mechanisms from the source language (like objects with private fields) are maintained at run time. This is a useful and widely used mechanism, but two important disadvantages are that (1) there is a performance penalty, as the bytecode is essentially interpreted or just-in-time compiled to real machine code, and (2) there is no protection against machine code attackers that can control machine code at lower layers of abstraction, for instance malware that has infected the operating system kernel is not constrained by the isolation mechanisms of the Java Virtual Machine.
- *Software Fault Isolation* [19] is an example of the class of *sandboxing* techniques, that make it possible for a trusted application to load untrusted binary modules into its address space. A critical assumption for these techniques is that the trusted application can inspect or even modify the untrusted module before it is loaded. By combinations of code analysis and code rewriting, the newly loaded module can be enforced not to do any harm. This technique is used for instance by web browsers to run native machine code as part of a web application [20]. But an important disadvantage is that these techniques are

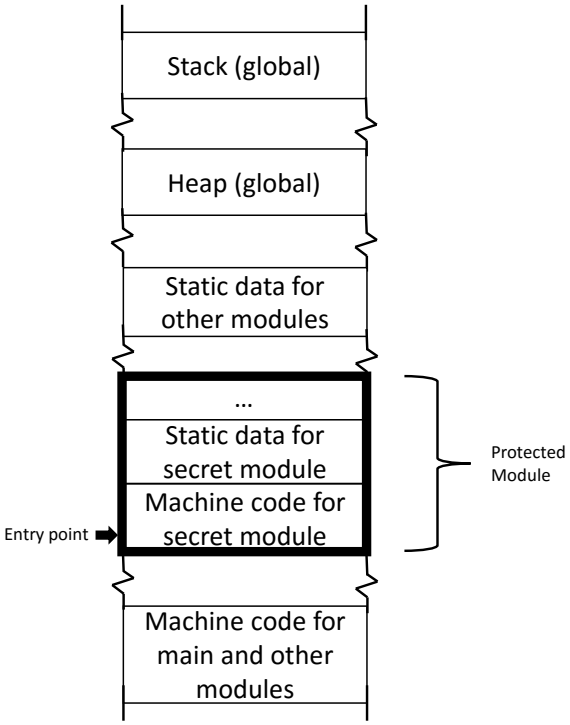

Fig. 3. A protected module

fundamentally asymmetric: they protect a host application from untrusted modules, but modules are not protected against the host application, and a fortiori not against malicious code in the operating system.

- *Capability machines* [21] provide an alternative memory protection mechanism at the level of the processor. Instead of using integers to address memory cells, the hardware supports *capabilities*, a kind of unforgeable pointer to memory segments that can also include limitations on what can be done to that memory. As a consequence, machine code is limited in what it can do by the capabilities it holds. Capability machines are a very powerful technique to achieve fine-grained separation of privileges at machine code level, and very recently progress has been made [22] in the formal characterization of capability safety that can lead to tool support for proving security properties of software on such machines. But an important downside is that they are still in a research stage, and no widely used hardware supports this model (yet).
- Finally, *Protected Module Architectures* [23], [24] offer a simple memory access control model that can be used to isolate modules. They have been designed both for higher-end processors [23], [24] as well as for small micro-processors [25], [26], [27], and recent Intel processors provide support under the name of Intel Software Guard Extensions (Intel SGX) [28].

secret.h

```
int get_secret(int get_pin())
```

secret.c

```
static int tries_left = 3;
static int PIN = 1234;
static int secret = 666;

int get_secret(int get_pin()) {
 if (tries_left > 0) {
   if (PIN == get_pin()) {
      tries_left = 3;
     return secret;}
   else { tries_left-- ; return 0; }; }
else return 0; }
```

Fig. 4. An alternative secret module

For the purpose of this paper, we will zoom in on Protected Module Architectures. Figure 3 shows how the secret module from the program shown in Figure 2 could be loaded in a protected module. A protected module is essentially a segment of memory, subdivided in a code part and a data part. In addition, the module has one or more *entry points*, addresses pointing into the code part of the protected module. The memory access control model of protected module architectures essentially enforces the following rules:

- When the Instruction Pointer (IP) is outside of the protected module, access to memory in the protected module is prohibited.
- When the IP is inside the protected module, data memory can be read and written, and code memory can be executed.
- The only way for the IP to enter a protected module is by jumping to one of the designated entry points.

This simple access control model makes it possible for modules to guard access to their private state. As shown in Figure 3, the secret module could be compiled such that its machine code goes into the code part of a protected module, and its static data goes in the data part. If we provide a single entry point to call the get_secret() function, then the variables PIN, tries_left and secret can only be accessed by the get_secret() function. Because of the memory access control, they can no longer be "scraped" from memory by malicious machine code in one of the other modules, or even by malware in the kernel.

B. Secure compilation

It is clear from the example discussed above, that the compiler will need to be modified to take into account the new protection mechanism offered by protected module architectures. And it turns out that making this compilation process secure is non-trivial. Here is a simple example of something that can go wrong.

Consider the variant of the secret module shown in Figure 4. Instead of passing in a candidate pin number, here clients of the module pass in a function pointer to a function get_pin() that the module will call to get a candidate pin from the user. This could be useful to allow clients to call the module with different implementations of get_pin(), for instance one that gets the candidate pin from standard input or another one that gets the candidate pin through a graphical user interface.

An important difference is that this new implementation accepts a function pointer as a parameter, whereas the old implementation accepted an integer. This difference is important: a malicious machine code client of the module can never provide an invalid integer (any 32-bit word is a valid integer), but it *can* provide invalid function pointers. Function pointers are represented at machine code level as 32-bit addresses, but only a few of these addresses actually point to the start of the code of a function.

What is even worse is that an attacker can actually exploit this to get his hands on the secret! For instance, the attacker could pass in the address of the machine code instruction within the compiled secret module that resets tries_left to 3 (i.e. the 4th line of the get_secret() function). When the secret module calls get_pin(), it will actually jump to the provided address, i.e. it will jump to the instruction that resets the tries_left variable and then returns secret as the result of get_pin(). The function get_secret() would then return 0 as the secret is probably not equal to the PIN, but the important thing is that the attacker has reset the tries_left variable! The attacker can use this exploit to reset the tries_left counter after each two tries, and hence can now successfully perform a brute force attack.

We can see from this example attack that the compiler will at least have to be very careful with arguments that outsiders can pass into a function that runs inside a protected module. For the example above, the compiler could for instance insert a defensive check that makes sure that the function pointer that is passed in should at least point to an address outside of the protected module.

An interesting question is: how can we ever know that defensive checks inserted by the compiler are sufficient? How can we know that we have thought of any possible exploit?

This question is the subject of current research in the area of *secure compilation*. Remember from the introduction that our security objective is: the compiled system should behave as specified in the source code that it is compiled from. The standard way of formalizing this requirement is to require the compiler to preserve and reflect observational equivalence [29]. In other words: whatever a machine code attacker can observe by interacting with a compiled module, could also be observed about that module by other source code modules interacting with the module. This is a precise and appropriate formalization of our security objective.

The question of how to securely compile C-like source code to protected module architectures has been the subject of several recent papers. Agten et al. [30] were the first to propose a secure compilation scheme, and Patrignani et al. [31] extended this scheme to handle many source code language features. The Sancus system [25] comes with a practical LLVM based compiler that implements a pragmatic variant of this secure compilation scheme.

Despite this progress, many interesting open questions remain. First, the work mentioned above focuses on compilation of a single protected module, and does not handle the case of multiple mutually distrustful modules. Extending the compiler to securely handle multiple modules is non-trivial and the subject of ongoing research [32], [33]. A second interesting question is how to deal with more advanced language features, in particular with stronger type systems. A stronger type system for the source code language can make it easier to reason about security properties at source code level, but it also makes it harder for a compiler to make sure that machine code attackers can only observe what (well-typed) source code modules could observe. Research on proving the security of compilations of typed languages is ongoing [34], [35].

C. Further extensions

Even if protected module architectures can use hardware-supported memory access control to make sure that even the operating system can not illegitimately read the state of a protected module, the operating system might still attack the module during loading of the module. One way to protect against such attacks is the use of remote attestation. The module will attest, after it has been loaded, that an unmodified version of the module is active in protected memory. The essence of the idea is to have the hardware derive a module-private cryptographic key that depends on the exact code that has been loaded (for instance by depending on a hash of the code segment of the loaded module). If the operating system were to modify the code of the module before loading it, the modified module would no longer receive the correct module-private key, and hence will fail to attest to remote parties that it has loaded correctly. Support for this kind of remote attestation of module authenticity is supported in several protected module architectures [27], [25], [28].

Finally, an important question is how such protected modules will receive their initial state, and will securely store and recover their state when they are stopped and restarted. For initialization of the module, the provider of the module can use the same module-private key that is used for remote attestation. But storing and recovering state should preferably be done locally on the device. Obviously the stored state should be confidentiality and integrity protected using cryptographic mechanisms, but in addition it should be secure against *rollback attacks* where an attacker tries to feed the module a stale stored state. Consider again the example program in Figure 2. In the initial state the tries_left variable will have the value 3. An example of a rollback attack would be that an attacker, after two unsuccessful tries, stops and restarts the module feeding it again the initial state which effectively resets the tries_left variable to 3 again, thus allowing the attacker to do a brute force attack against the PIN.

Supporting secure local storage and recovery of the states of protected modules turns out to be challenging, in particular if one wants to ensure *liveness* in the sense that random crashes or interruptions of the protected module should not leave it in a state where it can no longer make progress because none of the stored states is considered fresh anymore. Several designs have been proposed [36], [37], all of them imposing some additional hardware requirements.

V. Conclusions and outlook

The field of software security studies how software based systems can maintain desirable properties in the presence of intelligent attackers. It is useful to consider two aspects of this problem independently:

- An application-specific part: what are the desirable properties for the application at hand? How can users of the application misbehave? Software engineers should elicit these security requirement, propose a solid design with appropriate security countermeasures, and finally implement the desired application in source code taking care to avoid implementation vulnerabilities like the use of weak cryptography or incomplete mediation for access control.
- An application-independent but execution-platform-specific part: how can the execution platform ensure that the software based system will behave as specified in the source code, even in the presence of intelligent attackers that actively try to derail the software?

This paper has focused on the second aspect: how can one ensure that software that is compiled from source code to machine code on a specific execution platform can be guaranteed to behave as specified in the source code, and this for two kinds of attackers.

First, we discussed how memory safety vulnerabilities in unsafe languages like C allow attackers to derail software using an incredibly wide range of attack techniques for which the attacker only needs the ability to send input to and receive output from the software. We also surveyed part of the arsenal of countermeasures that C developers can rely on to make such attacks more difficult. The problem of protecting unsafe code against such I/O attackers is by now well understood, but such attacks still make up a significant fraction of the security incidents reported today.

Second, this paper also made the case that, in practice, even stronger attacker models should be considered. We discussed the model of the *machine code attacker*, that can execute arbitrary machine code in the virtual address space of the program under attack, or in the operating system. This is a more challenging problem, and research into adequate security mechanisms is still ongoing. We discussed some of the mechanisms that are already used in practice, and in particular zoomed in on the mechanism of Protected Module Architectures, supported in modern processors. Finally, we explained how compilers can use this mechanism to protect software against machine code attackers.

Acknowledgment

This work has been supported in part by the Intel Labs University Research Office. It is also partially supported by the Research Fund KU Leuven, and by the Research Foundation – Flanders (FWO).

References

[1] A. One, "Smashing the stack for fun and profit," *Phrack*, vol. 7, no. 49, November 1996.

[2] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security*, Washington, D.C.,, October 2007, pp. 552–561.

[3] N. Huq, "Pos ram scraper malware: Past, present, and future," Trend Micro, Tech. Rep., 2015.

[4] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.

[5] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter, "Breaking the memory secrecy assumption," in *EUROSEC*, 2009, pp. 1–8.

[6] U. Erlingsson, Y. Younan, and F. Piessens, "Low-level software security by example," in *Handbook of Information and Communication Security*. Springer, 2010.

[7] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, ser. SP '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 48–62.

[8] Y. Younan, W. Joosen, and F. Piessens, "Runtime countermeasures for code injection attacks against C and C++ programs," *ACM Comput. Surv.*, vol. 44, no. 3, pp. 17:1–17:28, Jun. 2012.

[9] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*. Berkeley, CA, USA: USENIX Association, 1998.

[10] N. D. Matsakis and F. S. Klock, II, "The Rust language," in *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, ser. HILT '14. New York, NY, USA: ACM, 2014.

[11] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of c," in *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, ser. ATEC '02. Berkeley, CA, USA: USENIX Association, 2002, pp. 275–288.

[12] R. C. Seacord, *The CERT C Secure Coding Standard*, 1st ed. Addison-Wesley Professional, 2008.

[13] B. Chess and J. West, *Secure Programming with Static Analysis*, 1st ed. Addison-Wesley Professional, 2007.

[14] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens, "VeriFast: A powerful, sound, predictable, fast verifier for C and Java," in *Proceedings of the Third International Conference on NASA Formal Methods*, ser. NFM'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 41–55.

[15] P. Agten, B. Jacobs, and F. Piessens, "Sound modular verification of C code executing in an unverified context," in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '15. New York, NY, USA: ACM, 2015, pp. 581–594.

[16] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-sanitizer: A fast address sanity checker," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 28–28.

[17] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and W. Joosen, "Paricheck: an efficient pointer arithmetic checker for C programs." in *ASIACCS*. ACM, 2010, pp. 145–156.

[18] T. Lindholm and F. Yellin, *Java Virtual Machine Specification*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[19] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," *SIGOPS Oper. Syst. Rev.*, vol. 27, no. 5, pp. 203–216, Dec. 1993. [Online]. Available: http://doi.acm.org/10.1145/173668.168635

[20] B. Yee, D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," in *IEEE Symposium on Security and Privacy (Oakland'09)*, IEEE, 3 Park Avenue, 17th Floor, New York, NY 10016, 2009.

[21] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The CHERI capability model: Revisiting RISC in an age of risk," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 457–468.

[22] D. Devriese, L. Birkedal, and F. Piessens, "Reasoning about object capabilities with logical relations and effect parametricity," ser. IEEE Euro S&P, Saarbrücken, Germany, 2016.

[23] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for TCB minimization," in *Proceedings of the ACM European Conference in Computer Systems (EuroSys)*. ACM, Apr. 2008, pp. 315–328.

[24] R. Strackx and F. Piessens, "Fides: Selectively hardening software application components against kernel-level or process-level malware," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 2–13.

[25] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. V. Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens, "Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base," in *22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX, 2013, pp. 479–498.

[26] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "Trustlite: A security architecture for tiny embedded devices," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys '14. New York, NY, USA: ACM, 2014, pp. 10:1–10:14.

[27] K. El Defrawy, A. Francillon, D. Perito, and G. Tsudik, "Smart: Secure and minimal architecture for (establishing a dynamic) root of trust," in *Proceedings of the Network & Distributed System Security Symposium (NDSS), San Diego, CA*, 2012.

[28] Intel, *Intel Software Guard Extensions Programming Reference*, 2014.

[29] M. Abadi, "Protection in programming-language translations," in *Secure Internet Programming, Security Issues for Mobile and Distributed Objects*, 1999, pp. 19–34.

[30] P. Agten, R. Strackx, B. Jacobs, and F. Piessens, "Secure compilation to modern processors," in *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, 2012, pp. 171–185.

[31] M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens, "Secure compilation to protected module architectures," *ACM Trans. Program. Lang. Syst.*, vol. 37, no. 2, pp. 6:1–6:50, Apr. 2015.

[32] M. Patrignani, D. Devriese, and F. Piessens, "Multi-module fully abstract compilation (extended abstract)," in *Workshop on Foundations of Computer Security*, July 2015.

[33] Y. Juglaret, C. Hriţcu, A. Azevedo de Amorim, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach, "Towards a fully abstract compiler using Micro-Policies: Secure compilation for mutually distrustful components," Technical Report, arXiv:1510.00697, 2015. [Online]. Available: http://arxiv.org/abs/1510.00697

[34] W. J. Bowman and A. Ahmed, "Noninterference for free," in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP 2015. New York, NY, USA: ACM, 2015, pp. 101–113.

[35] D. Devriese, M. Patrignani, and F. Piessens, "Fully-abstract compilation by approximate back-translation," in *POPL 2016*.

[36] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune, "Memoir: Practical state continuity for protected modules," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2011.

[37] R. Strackx, B. Jacobs, and F. Piessens, "Ice: A passive, high-speed, state-continuity scheme," in *Proceedings of the 30th Annual Computer Security Applications Conference*, ser. ACSAC '14. ACM, 2014.